

Reconstructing a Bounded-Degree Directed Tree Using Path Queries

Zhaosen Wang
CS, Purdue
West Lafayette, IN 47907, USA
wang2055@purdue.edu

Jean Honorio
CS, Purdue
West Lafayette, IN 47907, USA
jhonorio@purdue.edu

Abstract

We present a randomized algorithm for reconstructing directed rooted trees of n nodes and node degree at most d , by asking at most $\mathcal{O}(dn \log^2 n)$ *path queries*. Each path query takes as input an origin node and a target node, and answers whether there is a directed path from the origin to the target. Our algorithm is only a factor of $\mathcal{O}(d \log n)$ from the information-theoretic lower bound of $\Omega(n \log n)$ for any deterministic or randomized algorithm. We also present a $\mathcal{O}(dn \log^3 n)$ randomized algorithm for *noisy queries*, and a $\mathcal{O}(dn \log^2 n)$ randomized algorithm for *additive queries* on weighted trees.

1 Introduction

Scientists in diverse areas, such as statistics, epidemiology and economics, aim to unveil relationships within variables from collected data. This process can be seen as the task of reconstructing a graph (i.e., finding the hidden edges of a graph) by asking queries to an oracle. In this graph, each vertex is a variable, and each edge denotes the relationship between two variables. For instance, in cancer research, biologists try to discover the causal relationships between genes. By providing a specific treatment to a particular gene (origin), biologists can observe whether there is an effect in another gene (target). This effect can be either direct (if the two genes are connected with a directed edge) or indirect (if there is a directed path from the origin to the target gene.) In the above example, we can think of nature as the oracle that is answering whether there is a directed path from an origin to a target node. Arguably, the cost of asking queries is very high in several application domains. Thus, we are interested on the reconstruction of graphs that do not require the trivial n^2 queries for n nodes (i.e., one query for every possible pair of nodes.)

Prior work on reconstructing graphs has exclusively focused on *undirected* graphs. In [11], a $\mathcal{O}(dn \log n)$ algorithm was provided for recovering undirected trees of n nodes and maximum node degree d , by using queries that return the path length between two given nodes. Authors of [5] provide a $\mathcal{O}(dn)$ algorithm for reconstructing undirected trees of n nodes and maximum node degree d , and for a query that returns the distance from a given node to every other node. The results in [2, 3, 4] pertain to queries that answer whether there exists at least one edge between a given set of nodes. While [2, 3] focused on matchings and stars, the work of [4] provides a $\mathcal{O}(m \log n)$ algorithm for undirected graphs of n nodes and m edges. The results in [10, 14] pertain to queries that return the number of edges between a given set of nodes. A $\mathcal{O}(dn)$ algorithm was provided in [10] for undirected graphs of n nodes and maximum node degree d , while a $\mathcal{O}(m \log n)$ algorithm was given in [14] for undirected graphs of n nodes and m edges. Other works have focused on the recovery of *weighted* undirected graphs. The work of [9] provides a $\mathcal{O}(dn \log n)$ algorithm for recovering *weighted* undirected trees of n nodes and maximum node degree d , by using queries that return the sum of edge weights on the path between two given nodes. The work of [6, 13] pertains to the reconstruction of *weighted* undirected graphs of n nodes

and m edges. A $\mathcal{O}(m \log n)$ algorithm was provided for a query that gives the sum of edge weights between a given set of nodes.

The closest work to ours is [12], which provides a $\mathcal{O}(dn \log^2 n)$ randomized algorithm for *undirected* trees of n nodes and maximum node degree d , and for *separator queries*. A separator query takes three nodes i, k, j as input, and answers whether k is on the *undirected* path between i and j . In contrast, our work pertains to *directed rooted* trees. Furthermore, we use a different type of query which we call *path query*. A path query takes an ordered pair of nodes i, j as input, and answers whether there exists a directed path from i to j .

We provide a randomized algorithm for reconstructing directed rooted trees of n nodes and node degree at most d , in $\mathcal{O}(dn \log^2 n)$ time. To the best of our knowledge, there is no simple reduction to transform our problem to the problem of [12] or any of the above mentioned literature. Our algorithm relies on the *divide and conquer* approach, the use of *even separators* [7] and sorting. Regarding lower bounds, we show that any deterministic or randomized algorithm requires at least $\Omega(n \log n)$ queries. We also present a $\mathcal{O}(dn \log^3 n)$ randomized algorithm for a *noisy* regime, in which the bit that represents the oracles answer gets flipped with some probability, by an adversary, before it is revealed to the algorithm. Furthermore, we present a $\mathcal{O}(dn \log^2 n)$ randomized algorithm for reconstructing weighted trees by using *additive* queries that return the sum of the edge weights on the directed path between two given nodes. We finish the paper by showing some negative results that provide some motivation for our assumptions. We show that any deterministic or randomized algorithm requires at least $\Omega(n^2)$ queries in order to recover a general directed acyclic graph. We also show that any deterministic algorithm requires at least $\Omega(n^2)$ queries for recovering a family of sparse disconnected graphs, as well as a family of sparse connected graphs.

2 Preliminaries

In this section, we provide several formal definitions which will be useful later for the detailed description of our algorithm. For clarity, we also provide some preliminary introduction to the main aspects of our algorithm.

Let $G = (V, E)$ be a directed acyclic graph with vertex set V and edge set E . For clarity, when G is a directed rooted tree, we will use T instead of G . In this paper, we assume that T has n nodes, i.e., $|V| = n$. Furthermore, we also assume that the node degree is at most d . (In a directed acyclic graph, the node degree is the sum of the indegree and the outdegree of the node.)

Recall that a path in G from node i to node j (both in V) is a sequence of nodes $i, x_1, x_2, \dots, x_k, j$ such that $\{(i, x_1), (x_1, x_2), \dots, (x_{k-1}, x_k), (x_k, j)\}$ is a subset of the edge set E .

Our algorithm reconstructs a directed rooted tree, by using *path queries*. Next, we formally define path queries.

Definition 1. Let $G = (V, E)$ be a directed acyclic graph. A path query is a function $Q_G : V \times V \rightarrow \{0, 1\}$ such that $Q_G(i, j) = 1$ if there exists a path in G from i to j , and $Q_G(i, j) = 0$ otherwise.

Note that the above query only reveals a single bit of information, and it does not provide any information regarding the length of the path, thus making graph reconstruction a nontrivial task.

In this paper, we assume that the node set V is known, while edge set E is unknown. Our main problem is indeed to reconstruct E by using path queries. We will use $Q(i, j)$ to denote $Q_T(i, j)$ since for our problem, the directed rooted tree T is fixed (but unknown).

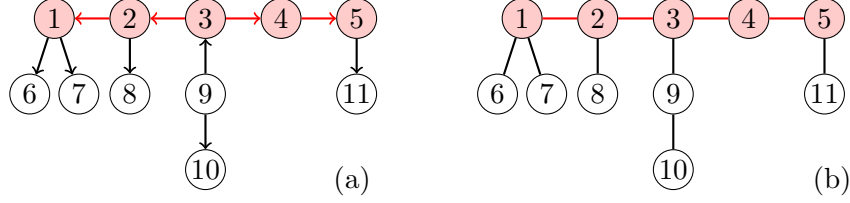


Figure 1: (a) A directed rooted tree with *multidirectional path* of nodes 1, 2, 3, 4, 5. Nodes and edges in the multidirectional path are shown in red. (b) Skeleton graph of the directed rooted tree on the left, with path of nodes 1, 2, 3, 4, 5. Nodes and edges in the path are shown in red.

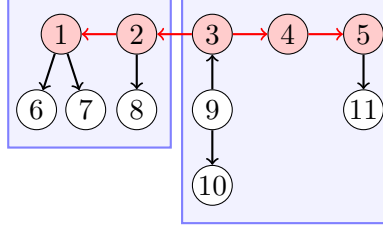


Figure 2: A directed rooted tree with multidirectional path of nodes 1, 2, 3, 4, 5. Nodes and edges in the multidirectional path are shown in red. The edge (2,3) in the multidirectional path is an *even separator*. Our algorithm will recursively recover each of the two generated subtrees (shown as blue boxes.)

A key step in our algorithm is the recovery of what we call *multidirectional paths*. A multidirectional path consists of the *directed* edges associated to an *undirected* path in the skeleton graph (i.e., the graph obtained by replacing each directed edge with an undirected edge in the original graph.) Figure 1 provides a visual illustration for intuitive understanding. Next, we formally define multidirectional paths.

Definition 2. Let $G = (V, E)$ be a directed acyclic graph. A multidirectional path of G between i and j is a sequence of nodes $i, x_1, x_2, \dots, x_{k-1}, x_k, j$ such that each node in V appears at most once in the sequence, and that there is an edge on either direction between each pair of adjacent nodes in the sequence. That is, either $(i, x_1) \in E$ or $(x_1, i) \in E$, either $(x_1, x_2) \in E$ or $(x_2, x_1) \in E$, \dots either $(x_{k-1}, x_k) \in E$ or $(x_k, x_{k-1}) \in E$, and either $(x_k, j) \in E$ or $(j, x_k) \in E$.

Next, we show that for directed rooted trees, a multidirectional path between any two arbitrary nodes always exists and is unique. More importantly, we show that a *multidirectional* path is either a *directed* path, or two *directed* paths that share the same origin (i.e., the *lowest common ancestor*.) Later, we leverage this property for recovering multidirectional paths (See Algorithm 2.)

Lemma 1. Let $T = (V, E)$ be a directed rooted tree. Given any two arbitrary nodes i and j , a multidirectional path of T between i and j always exists and is unique. Furthermore, a multidirectional path of T between i and j , is either a path from i to j , or a path from j to i , or two paths (one from k to i , and one from k to j , for some $k \in V - \{i, j\}$.) In this case, node k is the lowest common ancestor of i and j .

(See Appendix A for detailed proofs.)

Our algorithm relies on the divide and conquer approach. In order to apply the above approach in our problem, it is important to introduce the concepts of *even separators* and *bags*. Next, we introduce even separators.

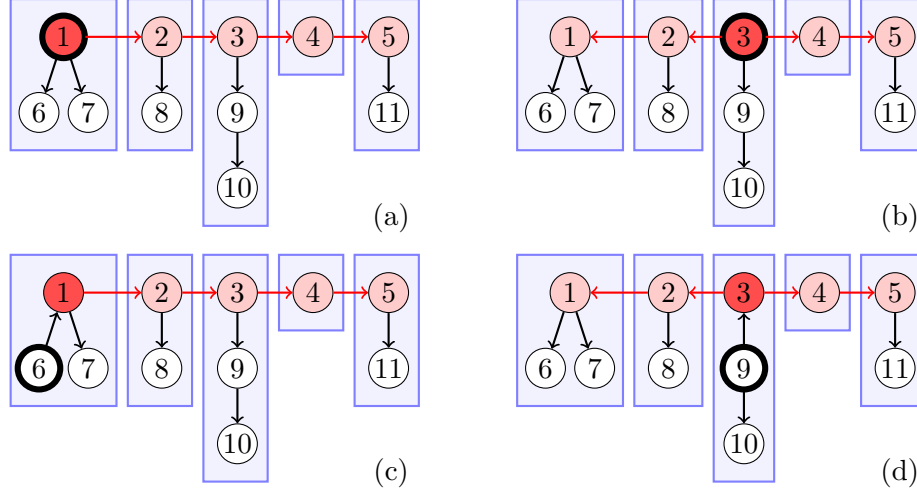


Figure 3: Four different directed rooted trees, and their *bags* with respect to the multidirectional path of nodes 1, 2, 3, 4, 5. Nodes and edges in the multidirectional path are shown in red. The lowest common ancestor of nodes 1 and 5 in the multidirectional path is shown in darker red. A tree root is shown as a thick circle. Bags are shown as blue boxes. Note that a multidirectional path can be a path (a,c) or two paths (b,d). The root can be in the multidirectional path (a,b). Otherwise, the root can be an ancestor of a node in the multidirectional path (c,d).

Definition 3. Let $T = (V, E)$ be a directed rooted tree of bounded degree d and let $n = |V|$. An even separator of T is an edge $e \in E$ that when removed from T , divides T into two subtrees T_1 and T_2 , where each of the subtrees have a number of nodes between n/d and $(d - 1)n/d$.

The existence of even separators is pivotal for using divide and conquer in our problem. Corollary 2.3 in [7] shows that if a graph is a bounded-degree directed tree, then an even separator exists. For our graph reconstruction problem, once the even separator is identified, we cut the tree through the even separator. This operation splits the tree into two subtrees. We then recursively call the algorithm for both subtrees. We illustrate this in Figure 2.

While even separators exist [7], it remains to know whether they can be efficiently found. We show later that (on average) there is an even separator *in the multidirectional path* between two nodes chosen independently and uniformly at random (See Theorem 1.)

In what follows, we formally define *bags*, which are also important for the divide and conquer approach taken here.

Definition 4. Let $T = (V, E)$ be a directed rooted tree, and S be the set of edges in a multidirectional path of T . Define $T_S = (V, E - S)$ as the subgraph of T after we remove all edges in S . A bag with respect to a node i in a multidirectional path with edges S , is a subset of nodes in V that contains i and all the nodes that are reachable from i in the (undirected) skeleton graph of T_S .

Intuitively speaking, we can think of edges in a tree as “ropes”. If we “nail” all nodes of a multidirectional path into the “wall”, then all other nodes will “hang” from one of the nodes in the path. Nodes that hang from the same particular node belong to the same bag. We include a visual example in Figure 3.

In our algorithm, we recover *exactly* all the directed edges *in a multidirectional path*. Bags are used to count the number of nodes associated to each node in the multidirectional path (without the need to recover all the directed edges.) For each edge in the multidirectional path, one can

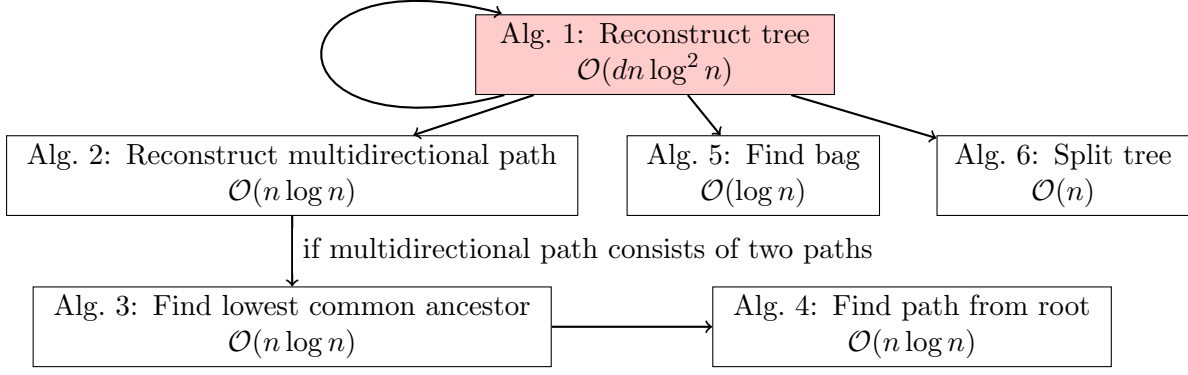


Figure 4: Main algorithm (in red), subroutines, and their time complexity.

then count the number of nodes on the two subtrees that would be generated if we were to cut the tree through the given edge. This process is used for identifying even separators.

Finally, our algorithm also performs sorting of nodes with a properly defined order relation, which is used for instance in the recovery of the directed edges in multidirectional paths.

Definition 5. Define the order relation of two nodes i and j as follows. If $Q(i, j) = 1$ we say that i is “less than” j , and “greater than” otherwise.

3 Algorithm

In this section, we present our randomized algorithm and analyze its time complexity. Our algorithm is similar in spirit to [12] which applies to *undirected* trees and *separator queries*. In this paper, we focus on *directed rooted* trees and *path queries*. We remind the reader that path queries only reveal a single bit of information, and they do not provide any information regarding the length of the path. (We discuss noisy and additive extensions in Section 4.)

In what follows, we explain our divide and conquer approach in our main Algorithm 1. A high-level overview is shown in Figure 4. First, we randomly pick two nodes and recover the sequence of nodes in the *multidirectional path* between those two randomly chosen nodes (See Algorithm 2.) Then, we divide the rest of the nodes (not in the multidirectional path) into *bags* defined by the multidirectional path (See Algorithm 5.) Later, we count the number of nodes on each bag, and determine if there exists an even separator in the multidirectional path. We repeat the whole procedure (i.e., from choosing a new pair of random nodes) until we find an even separator. Finally, we cut the tree through the even separator, thus effectively splitting the tree into two subtrees (See Algorithm 6.) We then recursively call our Algorithm 1 for both subtrees (until the input tree contains a single node.) On a more technical side, reconstructing a multidirectional path that consists of two paths that share the same origin, is more involved than reconstructing single paths. The former requires finding the lowest common ancestor of the two randomly chosen nodes (See Algorithm 3) as well as finding the root of the tree (See Algorithm 4.)

Next, we present our main Algorithm 1 for reconstructing a directed rooted tree by using path queries.

Algorithm 1 Reconstruct tree

```

1: Input: vertex set  $V$  of the directed rooted tree  $T$ 
2: if  $|V| = 1$  then
3:    $E \leftarrow \emptyset$ 
4:   break
5: end if
6: while true do
7:   Pick  $i, j \in V$  independently and uniformly at random
8:    $(\text{multidirpath}, \text{lowestancestorid}) \leftarrow \text{Reconstruct multidirectional path}(V, i, j)$ 
9:   Assume  $\text{multidirpath} = [x_1, \dots, x_P]$ 
10:   $\text{pathleft} \leftarrow \text{reverse}(\text{multidirpath}[1 \text{ to } \text{lowestancestorid}])$ 
11:   $\text{pathright} \leftarrow \text{multidirpath}[\text{lowestancestorid} \text{ to } P]$ 
12:   $\text{bagsize} \leftarrow [1, \dots, 1]$ , an array of size  $P$ 
13:  for each node  $k$  that is not on  $\text{multidirpath}$  do
14:     $\text{bagidxleft} \leftarrow \text{Find bag}(\text{pathleft}, k)$ 
15:     $\text{bagidxright} \leftarrow \text{Find bag}(\text{pathright}, k)$ 
16:    if  $\text{bagidxleft} = \text{lowestancestorid}$  then
17:       $\text{bagidx} \leftarrow \text{bagidxright} + \text{length}(\text{pathleft}) - 1$ 
18:    else
19:       $\text{bagidx} \leftarrow \text{bagidxleft}$ 
20:    end if
21:     $\text{bagsize}[\text{bagidx}] \leftarrow \text{bagsize}[\text{bagidx}] + 1$ 
22:  end for
23:   $\text{leftsize} \leftarrow 0$ 
24:   $\text{evenseparator} \leftarrow \text{None}$ 
25:  for  $r = 1, \dots, P - 1$  do
26:     $\text{leftsize} \leftarrow \text{leftsize} + \text{bagsize}[r]$ 
27:    if  $\text{leftsize} \in [n/d, (d-1)n/d]$  then
28:       $\text{evenseparator} \leftarrow (x_r, x_{r+1})$ 
29:      break
30:    end if
31:  end for
32:  if  $\text{evenseparator} \neq \text{None}$  then
33:    break
34:  end if
35: end while
36:  $(V_1, V_2) \leftarrow \text{Split tree}(V, \text{evenseparator})$ 
37:  $E_1 \leftarrow \text{Reconstruct tree}(V_1)$ 
38:  $E_2 \leftarrow \text{Reconstruct tree}(V_2)$ 
39:  $E \leftarrow E_1 \cup E_2 \cup \{\text{evenseparator}\}$ 
40: Output: edge set  $E$ 

```

We now clarify some additional details regarding Algorithm 1. Finding out the bag of a particular node with respect to a *directed* path is relatively easier than with respect to a *multidirectional* path. Note that by Lemma 1, a *multidirectional* path is either a *directed* path, or two *directed* paths that share the same origin (i.e., the lowest common ancestor.) In Algorithm 1, we simplify the bag assignment task, by splitting a *multidirected* path (that is not a single path) into its two constituent *directed* paths. Lines 8-11 first recover the lowest common ancestor, and then breaks the multidirectional path into two paths. For instance, the multidirectional path 1, 2, 3, 4, 5 in Figure 3(d) with lowest common ancestor 3 would become two paths 3, 2, 1 and 3, 4, 5.

Next, we explain each subroutine used in our main Algorithm 1. The first key step is to recover the multidirectional path between two randomly chosen nodes. Next, we present Algorithm 2 for recovering the sequence of nodes in the multidirectional path between any two nodes.

The idea behind Algorithm 2 is as follows. Let $i, j \in V$ be two arbitrary nodes in the directed rooted tree $T = (V, E)$. Recall that some *multidirectional* paths are a single *directed* path. We can easily detect this case by asking whether there is a path from i to j (i.e., whether $Q(i, j) = 1$), or whether there is a path from j to i (i.e., whether $Q(j, i) = 1$.) Without loss of generalization, assume there is a path in the directed rooted tree T from i to j . (We could similarly assume that there is a path from j to i .) We can find out the set of all nodes $\{x_1, x_2, \dots, x_k\}$ on the path from i to j , by using path queries. For all nodes $k \in V - \{i, j\}$, we ask the oracle about $Q(i, k)$ and $Q(k, j)$. Note that k is on the path from i to j , if and only if $Q(i, k) = Q(k, j) = 1$. After finding out the set of all nodes $\{x_1, x_2, \dots, x_k\}$ on the path from i to j , it remains to sort the nodes in order to obtain the correct sequence, thus recovering the path. We sort the list of nodes $\{i, x_1, x_2, \dots, x_k, j\}$ by using the order relation given in Definition 5.

Some *multidirectional* paths consist of two *directed* paths that share the same origin m . In this case, Algorithm 2 first recovers the lowest common ancestor m and then reconstructs two *directed* paths: one from m to i , and one from m to j , by following the approach explained before.

Next, we present Algorithm 2 in detail.

Algorithm 2 Reconstruct multidirectional path

```
1: Input: vertex set  $V$  of the directed rooted tree  $T$ , two nodes  $i, j \in V$ 
2: multidirpath  $\leftarrow []$ , lowestancestoridx  $\leftarrow \text{None}$ 
3: if  $Q(i, j) = 1$  then
4:   for each node  $k \in V - \{i, j\}$  where  $Q(i, k) = 1$  and  $Q(k, j) = 1$  do
5:     multidirpath  $\leftarrow \text{append}(\text{multidirpath}, k)$ 
6:   end for
7:   Sort multidirpath with the order relation given in Definition 5
8:   multidirpath  $\leftarrow \text{append}(i, \text{multidirpath}, j)$ 
9:   lowestancestoridx  $\leftarrow 1$ 
10: else if  $Q(j, i) = 1$  then
11:   for each node  $k \in V - \{i, j\}$  where  $Q(j, k) = 1$  and  $Q(k, i) = 1$  do
12:     multidirpath  $\leftarrow \text{append}(\text{multidirpath}, k)$ 
13:   end for
14:   Sort multidirpath with the order relation given in Definition 5
15:   multidirpath  $\leftarrow \text{append}(j, \text{multidirpath}, i)$ 
16:   lowestancestoridx  $\leftarrow 1$ 
17: else
18:    $m \leftarrow \text{Find lowest common ancestor}(V, i, j)$ 
19:   pathleft  $\leftarrow []$ , pathright  $\leftarrow []$ 
20:   for each node  $k \in V - \{m, i\}$  where  $Q(m, k) = 1$  and  $Q(k, i) = 1$  do
21:     pathleft  $\leftarrow \text{append}(\text{pathleft}, k)$ 
22:   end for
23:   Sort pathleft with the order relation given in Definition 5
24:   for each node  $k \in V - \{m, j\}$  where  $Q(m, k) = 1$  and  $Q(k, j) = 1$  do
25:     pathright  $\leftarrow \text{append}(\text{pathright}, k)$ 
26:   end for
27:   Sort pathright with the order relation given in Definition 5
28:   multidirpath  $\leftarrow \text{append}(i, \text{reverse}(\text{pathleft}), m, \text{pathright}, j)$ 
29:   lowestancestoridx  $\leftarrow 2 + \text{length}(\text{pathleft})$ 
30: end if
31: Output: multidirpath, lowestancestoridx
```

As seen in Algorithm 2, reconstructing a multidirectional path that consists of two paths requires finding the lowest common ancestor of the two randomly chosen nodes. Our Algorithm 3 finds the lowest common ancestor of a multidirectional path between any two arbitrary nodes.

Algorithm 3 works as follows. Let $i, j \in V$ be two arbitrary nodes of the directed rooted tree $T = (V, E)$. First, we recover the *directed* path from the root to i . We assume that the order of the nodes in the above path follow the order relation given in Definition 5. Thus, the tree root is the first element on such path. Then, we iterate through all nodes in the path, in order to find the last ancestor of j in the path from the root. This last ancestor is indeed the lowest common ancestor of i and j .

Next, we present Algorithm 3 for finding the lowest common ancestor of a multidirectional path between any two arbitrary nodes.

Algorithm 3 Find lowest common ancestor

```
1: Input: vertex set  $V$  of the directed rooted tree  $T$ , two nodes  $i, j \in V$ 
2: pathfromroot  $\leftarrow \text{Find path from root}(V, i)$ 
3: for each node  $k$  in pathfromroot do
4:   if  $Q(k, j) = 1$  then
5:     lowestcommonancestor  $\leftarrow k$ 
6:   break
7: end if
8: end for
9: Output: lowestcommonancestor
```

As seen in Algorithm 3, in order to find the lowest common ancestor, one has to find the root of the directed tree. Our Algorithm 4 identifies the path from the root to a given arbitrary node.

The inner workings of Algorithm 4 are as follows. Let $i \in V$ be an arbitrary node in the directed rooted tree $T = (V, E)$. For each node $j \in V - \{i\}$, we ask the oracle about $Q(j, i)$. If $Q(j, i) = 1$ then there is a path from j to i , and therefore we add node j to the list of nodes that reach i . In order to recover the *directed* path from the root to i , we sort the aforementioned list

of nodes, by using the order relation given in Definition 5. That is, the first element on the sorted list is the tree root.

Next, we present Algorithm 4 in detail.

Algorithm 4 Find path from root

```

1: Input: vertex set  $V$  of the directed rooted tree  $T$ , node  $i \in V$ 
2:  $\text{pathfromroot} \leftarrow []$ 
3: for each node  $j \in V - \{i\}$  do
4:   if  $Q(j, i) = 1$  then
5:      $\text{pathfromroot} \leftarrow \text{append}(\text{pathfromroot}, j)$ 
6:   end if
7: end for
8: Sort  $\text{pathfromroot}$  with the order relation given in Definition 5
9: Output:  $\text{pathfromroot}$ 

```

The second key step in our main Algorithm 1 is to divide the nodes (which are not in the multidirectional path) into bags defined by the multidirectional path. As argued before, if the *multidirectional* path consists of two *directed* paths, then Algorithm 1 breaks the *multidirectional* path into its two constituent *directed* paths, by using the least common ancestor. Thus, the bag assignment task needs only to consider directed paths, as we do in Algorithm 5.

Here we give an intuitive explanation of the bag assignment task in Algorithm 5. For instance, consider finding out the bag of node 10 in the directed path 1, 2, 3, 4, 5 in Figure 3(a). We see that nodes 1, 2 and 3 are all ancestors of node 10 (i.e., $Q(1, 10) = Q(2, 10) = Q(3, 10) = 1$.) We also see that nodes 4 and 5 are not ancestors of node 10 (i.e., $Q(4, 10) = Q(5, 10) = 0$.) Note that node 10 belongs to the bag of node 3. The above suggests that the task of finding out the bag of node 10 can be done by searching for a node i for which $Q(i, 10) = 1$ and $Q(j, 10) = 0$ where (i, j) is an edge in the path. This can be efficiently done by performing binary search.

There are two exceptions to the above rule: when queries for all nodes in the path return 1, and when queries for all nodes in the path return 0. As an example for the first case (when all queries return 1), consider finding out the bag of node 11 in the directed path 1, 2, 3, 4, 5 in Figure 3(a). We see that nodes 1, 2, 3, 4 and 5 are all ancestors of node 11 (i.e., $Q(1, 11) = Q(2, 11) = Q(3, 11) = Q(4, 11) = Q(5, 11) = 1$.) In this case, we assign node 11 to the bag of the last node in the path, i.e., 5. As an example for the second case (when all queries return 0), consider finding out the bag of node 10 in the directed path 3, 4, 5 in Figure 3(d). We see that neither nodes 3, 4 or 5 are ancestors of node 10 (i.e., $Q(3, 10) = Q(4, 10) = Q(5, 10) = 0$.) In this case, we assign node 10 to the bag of the lowest common ancestor, i.e., 3.

Fortunately, all the cases analyzed above are naturally handled by binary search. Next, we present Algorithm 5 for finding out the bag of a node with respect to an arbitrary directed path.

Algorithm 5 Find bag

```

1: Input: path  $x_1, x_2, \dots, x_k$ , node  $i \in V$  where  $V$  is the vertex set of the directed rooted tree  $T$ 
2:  $l \leftarrow 1, r \leftarrow k$ 
3: while  $l < r$  do
4:    $m \leftarrow (l + r)/2$ 
5:   if  $Q(x_m, i) = 1$  then
6:      $l \leftarrow m$ 
7:   else
8:      $r \leftarrow m$ 
9:   end if
10: end while
11: Output: bag index  $l \in \{1, \dots, k\}$ 

```

The final step in our main Algorithm 1 is to cut the tree through the even separator, which splits the tree into two subtrees. Our Algorithm 6 splits a directed rooted tree into two subtrees, by cutting the original tree through any arbitrary edge.

The idea behind Algorithm 6 is as follows. Let $e = (i, j) \in E$ be an arbitrary edge in the directed rooted tree $T = (V, E)$. Let T_1 and T_2 be two subtrees that result from removing e from T . Note that every node must belong to either T_1 or T_2 . Without loss of generality, let $i \in T_1$ and $j \in T_2$. Since T is a directed rooted tree, j has one parent in T , which is indeed i . Removing (i, j) from T makes j have no parents in T_2 . Therefore, j is the root of T_2 . For all nodes $k \in V - \{j\}$, we ask the oracle about $Q(j, k)$. If $Q(j, k) = 1$, then k belongs to T_2 , otherwise k belongs to T_1 .

Next, we present Algorithm 6 in detail.

Algorithm 6 Split tree

```

1: Input: vertex set  $V$  of the directed rooted tree  $T = (V, E)$ , edge  $(i, j) \in E$ 
2:  $V_1 \leftarrow \emptyset, V_2 \leftarrow \{j\}$ 
3: for each node  $k \in V - \{j\}$  do
4:   if  $Q(j, k) = 1$  then
5:      $V_2 \leftarrow V_2 \cup \{k\}$ 
6:   else
7:      $V_1 \leftarrow V_1 \cup \{k\}$ 
8:   end if
9: end for
10: Output: partitions  $V_1, V_2$  of  $V$ 

```

We finish the section by analyzing the time complexity of our randomized algorithm.

Theorem 1. *Algorithm 1 takes $\mathcal{O}(dn \log^2 n)$ expected time, in order to reconstruct a directed rooted tree of n nodes and node degree at most d . Furthermore, for a fixed probability of error $\delta \in (0, 1)$, Algorithm 1 takes at most $\mathcal{O}(\frac{1}{\delta} dn \log^2 n)$ time, with probability at least $1 - \delta$.*

(See Appendix A for detailed proofs.)

4 Lower Bound and Extensions

In this section, we study lower bounds for reconstructing directed rooted trees from path queries. We also extend our original algorithm for the case of noisy queries, as well as the reconstruction of weighted trees from additive queries. Finally, we provide negative results for directed acyclic graphs that provide some motivation for our assumptions.

4.1 Lower Bound

Here, we present the information-theoretic lower bound for reconstructing directed rooted trees from path queries. Interestingly, the following result does not depend on the node degree d .

Theorem 2. *In order to reconstruct a directed rooted tree of n nodes and node degree at most d , any deterministic algorithm requires at least $\Omega(n \log n)$ time. Furthermore, any randomized algorithm requires at least $\Omega((1 - \delta)n \log n)$, otherwise it would fail with probability at least δ .*

Given the above and Theorem 1, our algorithm is only a factor of $\mathcal{O}(d \log n)$ from the information-theoretic limit.

4.2 Noisy Queries

Here, we analyze a *noisy* regime, in which the bit that represents the oracle's answer gets flipped with some probability, by an adversary, before it is revealed to the algorithm. Next, we formally define noisy queries.

Definition 6. Let $G = (V, E)$ be a directed acyclic graph, and let Q_G be a path query. A noisy path query with noise parameter $\varepsilon \in (0, 1/2)$ is a function $\tilde{Q}_G : V \times V \rightarrow \{0, 1\}$ such that $\tilde{Q}_G(i, j) = Q_G(i, j)$ with probability $1 - \varepsilon$, and $\tilde{Q}_G(i, j) = 1 - Q_G(i, j)$ with probability ε .

In order to make use of our original algorithm, we proceed with the following strategy. For each node pair, we will perform majority voting on m noisy path queries. If m is large enough, noise will be removed with high probability.

Next, we present Algorithm 7 for reconstructing a directed rooted tree by using *noisy* path queries. We will use $\tilde{Q}^{(k)}$ to denote the k -th call to the query $\tilde{Q}_T(i, j)$, since for our problem, the directed rooted tree T is fixed (but unknown).

Algorithm 7 Reconstruct tree from noisy queries

- 1: **Input:** vertex set V of the directed rooted tree T , number of queries m per node pair
 - 2: Define the *path query* Q' as follows. Let $Q'(i, j) = 1$ if $\sum_{k=1}^m \tilde{Q}^{(k)}(i, j) > m/2$, and $Q'(i, j) = 0$ otherwise.
 - 3: $E \leftarrow \text{Reconstruct tree}(V)$
 - 4: **Output:** edge set E
-

The above opens up a question on the number of queries m per node pair, that are sufficient to guarantee graph recovery success. A second question is whether m depends on the number of nodes n and maximum node degree d , thus affecting the time complexity of our randomized algorithm. The following theorem answers both questions.

Theorem 3. For a fixed probability of error $\delta \in (0, 1)$ and noise parameter $\varepsilon \in (0, 1/2)$, Algorithm 7 takes at most $\mathcal{O}(\frac{1}{\delta} \frac{1}{(1/2-\varepsilon)^2} dn \log^2 n (\log d + \log n + \log \frac{1}{\delta}))$ time, with probability at least $1 - \delta$, in order to reconstruct a directed rooted tree of n nodes and node degree at most d , provided that the number of queries per node pair fulfills $m \in \Theta(\frac{1}{(1/2-\varepsilon)^2} (\log d + \log n + \log \frac{1}{\delta}))$.

In practice we do not need to know the exact value of the noise parameter ε . A lower bound of ε suffices to define the number of queries m per node pair.

4.3 Additive Queries on Weighted Trees

Here, we focus on the reconstruction of *weighted* directed rooted trees by using additive queries. An additive path query returns the sum of the edge weights on the directed path between two given nodes, if such a path exists, or zero otherwise. Next, we formally define additive path queries.

Definition 7. Let $T = (V, E, W)$ be a weighted directed rooted tree, with positive weights for each edge, i.e., $w_{i,j} > 0$ for all $(i, j) \in E$, and $w_{i,j} = 0$ for all $(i, j) \notin E$. A additive path query is a function $\hat{Q}_T : V \times V \rightarrow [0, +\infty)$ such that if there exists a path in T from i to j then $\hat{Q}_T(i, j)$ is the sum of the weights of the edges in the path, and $\hat{Q}_T(i, j) = 0$ otherwise.

Note that the above query reveals much more information compared to the path query in Definition 1 which only reveals a single bit of information. Our strategy is to convert the *additive*

query problem into our original problem for recovering the edge set. Afterwards, we recover the edge weights by calling the *additive* queries for each edge.

Next, we present Algorithm 8 for reconstructing a weighted directed rooted tree by using *additive* path queries. We will use $\widehat{Q}(i, j)$ to denote $\widehat{Q}_T(i, j)$ since for our problem, the weighted directed rooted tree T is fixed (but unknown).

Algorithm 8 Reconstruct weighted tree from additive queries

```

1: Input: vertex set  $V$  of the weighted directed rooted tree  $T$ 
2: Define the path query  $Q$  as follows. Let  $Q(i, j) = 1$  if  $\widehat{Q}(i, j) > 0$ , and  $Q(i, j) = 0$  otherwise.
3:  $E \leftarrow \text{Reconstruct tree}(V)$ 
4:  $W \leftarrow 0$ 
5: for each edge  $(i, j) \in E$  do
6:    $w_{i,j} = \widehat{Q}(i, j)$ 
7: end for
8: Output: edge set  $E$ , edge weights  $W$ 

```

The time complexity of the above algorithm is as follows.

Theorem 4. *Algorithm 8 takes $\mathcal{O}(dn \log^2 n)$ expected time, in order to reconstruct a weighted directed rooted tree of n nodes and node degree at most d . Furthermore, for a fixed probability of error $\delta \in (0, 1)$, Algorithm 8 takes at most $\mathcal{O}(\frac{1}{\delta} dn \log^2 n)$ time, with probability at least $1 - \delta$.*

4.4 Negative Results for Directed Acyclic Graphs

As argued before, the cost of asking queries is very high in several application domains. Thus, we are interested on the reconstruction of graphs that do not require the trivial n^2 queries for n nodes (i.e., one query for every possible pair of nodes.) A natural question is whether a general directed acyclic graph could be recovered efficiently by asking less than $\Omega(n^2)$ queries. Next, we provide a negative answer for the above.

Theorem 5. *In order to reconstruct a directed acyclic graph of n nodes, any deterministic algorithm requires at least $\Omega(n^2)$ time. Furthermore, any randomized algorithm requires at least $\Omega((1 - \delta)n^2)$, otherwise it would fail with probability at least δ .*

Recall that our algorithm pertains to directed rooted trees with a maximum node degree. These graphs are not only sparse but also *weakly* connected (i.e., their undirected skeleton graphs are connected.) One could ask whether connectedness is a necessary condition, and whether sparsity makes graph reconstruction easier. Next, we show that an algorithm requires $\Omega(n^2)$ queries for recovering a family of sparse disconnected graphs, as well as a family of sparse connected graphs.

Theorem 6. *In order to reconstruct a sparse disconnected directed acyclic graph of n nodes, any deterministic algorithm requires at least $\Omega(n^2)$ time.*

Theorem 7. *In order to reconstruct a sparse connected directed acyclic graph of n nodes, any deterministic algorithm requires at least $\Omega(n^2)$ time.*

5 Concluding Remarks

There are several ways of extending this research. The analysis of the reconstruction of other families of graphs in $\mathcal{O}(n \log n)$ time would be of great interest. Given our results for directed rooted trees, it would be interesting to analyze other families of sparse connected graphs, such as graphs with bounded tree-width as well as graphs with bounded arboricity.

References

- [1] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.
- [2] Noga Alon and Vera Asodi. Learning a hidden subgraph. *International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science*, 3142:110–121, 2004.
- [3] Noga Alon, Richard Beigel, Simon Kasif, Steven Rudich, and Benny Sudakov. Learning a hidden matching. *SIAM Journal on Computing*, 33(2):487–501, 2004.
- [4] Dana Angluin and Jiang Chen. Learning a hidden graph using $O(\log n)$ queries per edge. *Conference on Learning Theory*, pages 210–223, 2004.
- [5] Zuzana Beerliova, Felix Eberhard, Thomas Erlebach, Alexander Hall, Michael Hoffmann, Matúš Mihal’ák, and L. Shankar Ram. Network discovery and verification. *IEEE Journal on Selected Areas in Communications*, 24(12):2168–2181, 2006.
- [6] Sung-Soon Choi. Polynomial time optimal query algorithms for finding graphs with arbitrary real weights. *Conference on Learning Theory*, pages 1–22, 2013.
- [7] Fan Chung. Separator theorems and their applications. *Paths, Flows, and VLSI-Layout*, pages 17–34, 1990.
- [8] T. Cover and J. Thomas. *Elements of Information Theory*. John Wiley & Sons, 2nd edition, 2006.
- [9] Joseph Culberson and Piotr Rudnicki. A fast algorithm for reconstructing trees from distance matrices. *Information Processing Letters*, 30(4):215–220, 1989.
- [10] Vladimir Grebinski and Gregory Kucherov. Optimal reconstruction of graphs under the additive model. *Algorithmica*, 28(1):104–124, 2000.
- [11] Jotun Hein. An optimal algorithm to reconstruct trees from additive distance data. *Bulletin of Mathematical Biology*, 51(5):597–603, 1989.
- [12] M. Jagadish and Anindya Sen. Learning a bounded-degree tree using separator queries. *Algorithmic Learning Theory, Lecture Notes in Computer Science*, 8139:188–202, 2013.
- [13] Hanna Mazzawi. Optimally reconstructing weighted graphs using queries. *ACM-SIAM Symposium on Discrete Algorithms*, pages 608–615, 2010.
- [14] Lev Reyzin and Nikhil Srivastava. Learning and verifying graphs using queries with a focus on edge counting. *Algorithmic Learning Theory, Lecture Notes in Computer Science*, 4754:285–297, 2007.

A Detailed Proofs

In this section, we state the proofs of all the theorems and claims in our manuscript.

A.1 Proof of Lemma 1

Proof. Existence and uniqueness follows straightforwardly from the fact that the directed rooted tree T is *weakly* connected (i.e., the undirected skeleton graph of T is connected.) For the second claim, note that each node in $T = (V, E)$ has at most one parent. Given any three different nodes $p, q, r \in V$, the following condition holds $\neg((p, r) \in E \wedge (q, r) \in E)$. Thus, we can only construct the three cases provided in the statement, otherwise we would violate the latter condition. \square

A.2 Proof of Theorem 1

Proof. Let a “round” be a repetition of the “while” loop at Line 6 of Algorithm 1. We show that there are $\mathcal{O}(d)$ rounds in expectation, and that each round takes $\mathcal{O}(n \log n)$ time. We then finish the proof by the application of the master theorem.

First, we analyze the expected number of rounds for finding an even separator *in the multidirectional path* between two nodes chosen independently and uniformly at random. Recall that by Lemma 1 for directed rooted trees, a multidirectional path between any two arbitrary nodes always exists and is unique. Thus, it remains to analyze the probability that two randomly chosen nodes lie on a different subtree defined by the even separator.

Recall that Corollary 2.3 in [7] shows if the tree T has node degree at most d , then an even separator exists. More specifically, from Definition 3, we know the even separator splits the tree into two subtrees, where each of the subtrees have a number of nodes between n/d and $(d-1)n/d$. Next, we reason about the two randomly selected nodes on each round. Let q be the proportion of nodes on the first subtree, and $1 - q$ be the proportion of nodes on the second subtree. We know that $q \in [1/d, (d-1)/d]$ and similarly $1 - q \in [1/d, (d-1)/d]$. Since both nodes are selected independently and uniformly at random from the set of n nodes, then q^2 is the probability that both nodes fall in the first subtree. Similarly, $(1 - q)^2$ is the probability that both nodes fall in the second subtree. The probability p that the two nodes lie on a different subtree is

$$\begin{aligned} p &\geq \min_{q \in [1/d, (d-1)/d]} (1 - q^2 - (1 - q)^2) \\ &= 1 - (1/d)^2 - ((d-1)/d)^2 \\ &= 2(d-1)/d^2 \\ &\in \Omega(1/d) . \end{aligned}$$

Therefore, the expected number of rounds r until we successfully find two nodes lying on a different subtree is $E[r] = \sum_{r=1}^{\infty} r(1-p)^{r-1}p = 1/p$ which is $\mathcal{O}(d)$. We now show that each round takes $\mathcal{O}(n \log n)$ time. First, we derive a sequence of conclusions regarding the subroutines:

- Algorithm 4 takes $\mathcal{O}(n \log n)$ time, since the most time-consuming step is sorting in Line 8.
- Algorithm 3 takes $\mathcal{O}(n \log n)$ time, since the most time-consuming step is the call to Algorithm 4 in Line 2, which takes $\mathcal{O}(n \log n)$ time.
- Algorithm 2 takes $\mathcal{O}(n \log n)$ time, since the most time-consuming steps are sorting in Lines 7, 14, 23 and 27, and the call to Algorithm 3 in Line 18, which takes $\mathcal{O}(n \log n)$ time.

- Algorithm 5 takes $\mathcal{O}(\log n)$ time, since it performs binary search.
- Algorithm 6 takes $\mathcal{O}(n)$ time, since the “for” loop in Line 3 iterates for at most n times.

Recall that each round calls Algorithms 2, 5 and 6. It can be observed then, that the most time-consuming step on each round is the call to Algorithm 2, which takes $\mathcal{O}(n \log n)$ time.

Thus, so far we know that the time complexity of the “while” loop at Line 6 is $\mathcal{O}(dn \log n)$. To finalize the proof, note that Algorithm 1 exits the “while” loop at Line 6 when it finds an even separator. Recall from Definition 3 that the even separator splits the tree into two subtrees, where each of the subtrees have a number of nodes between n/d and $(d-1)n/d$. The total running time for Algorithm 1 is given by the recursive formula

$$C(n) = C(n/d) + C((d-1)n/d) + \mathcal{O}(dn \log n) .$$

For clarity, we rewrite $C(n)$ in terms of the master theorem in [1]. That is, $C(n) = \alpha_1 C(\beta_1 n) + \alpha_2 C(\beta_2 n) + \gamma(n)$, for $\alpha_1 = \alpha_2 = 1$, $\beta_1 = 1/d$, $\beta_2 = (d-1)/d$ and $\gamma(n) \in \mathcal{O}(dn \log n)$. By invoking the master theorem in [1], we have that $C(n) \in \mathcal{O}(n^s \int_1^n \gamma(z)/z^{s+1} dz)$, where s is the value for which $\alpha_1 \beta_1^s + \alpha_2 \beta_2^s = 1$. In our case $s = 1$ and thus $C(n) \in \mathcal{O}(dn \log^2 n)$.

For the remainder of the proof, we will use C to denote $C(n)$ since n is a constant. Note that C is a non-negative random variable with expectation $\mathbb{E}[C] \in \mathcal{O}(dn \log^2 n)$. By Markov’s inequality, we have that $\mathbb{P}[C > a] \leq \mathbb{E}[C]/a$. By letting $a = \mathbb{E}[C]/\delta$ we have $\mathbb{P}[C > \mathbb{E}[C]/\delta] \leq \delta$. Therefore $\mathbb{P}[C \leq \mathbb{E}[C]/\delta] \geq 1 - \delta$, and we prove our claim that $\mathbb{P}[C \in \mathcal{O}(\frac{1}{\delta} dn \log^2 n)] \geq 1 - \delta$. \square

A.3 Proof of Theorem 2

Proof. Let $\mathcal{T}_{n,d}$ be the set of directed rooted trees of n nodes and node degree at most d . Next, we show that $\log |\mathcal{T}_{n,d}| \in \Theta(n \log n)$. Interestingly, the latter (tight) bound does not depend on d . For a lower bound, note that the number of directed rooted trees with at most one children is equal to the number of permutations of n nodes, which is $n!$. We have $\log |\mathcal{T}_{n,d}| \geq \log n! \geq n(\log n - 1) \geq \frac{1}{4}n \log n$ for $n \geq 4$ and therefore $\log |\mathcal{T}_{n,d}| \in \Omega(n \log n)$. For an upper bound, note that the number of directed rooted trees with no constraint in the number of children is equal to the number of directed spanning trees, which is n^{n-1} by Cayley’s formula. We have $\log |\mathcal{T}_{n,d}| \leq \log n^{n-1} \leq n \log n$ and therefore $\log |\mathcal{T}_{n,d}| \in \mathcal{O}(n \log n)$. From the above, we conclude that $\log |\mathcal{T}_{n,d}| \in \Theta(n \log n)$.

Since each query only reveals a single bit of information, the lower bound of $\Omega(n \log n)$ for *deterministic* algorithms follows.

Now we provide an information-theoretic lower bound for *randomized* algorithms. Assume that nature picks a directed rooted tree $T^* \in \mathcal{T}_{n,d}$ uniformly at random. Any mechanism for finding out the correct tree is allowed to make C possibly-dependent queries (for C node pairs) for which the binary responses are Q_1, \dots, Q_C . Let $T \in \mathcal{T}_{n,d}$ be the directed rooted tree that is guessed on the basis of the C query responses. The above defines a Markov chain $T^* \rightarrow (Q_1, \dots, Q_C) \rightarrow T$.

By properties of the mutual information, and since the joint responses (Q_1, \dots, Q_C) can take up to 2^C possible values, we have:

$$\begin{aligned} \mathbb{I}(T^*; Q_1, \dots, Q_C) &\leq \mathbb{H}(Q_1, \dots, Q_C) \\ &\leq C \log 2 . \end{aligned}$$

By the Fano's inequality [8] on the Markov chain $T^* \rightarrow (Q_1, \dots, Q_C) \rightarrow T$ we have

$$\begin{aligned} \mathbb{P}[T \neq T^*] &\geq 1 - \frac{\mathbb{I}(T^*; Q_1, \dots, Q_C) + \log 2}{\log |\mathcal{T}_{n,d}|} \\ &\geq 1 - \frac{C \log 2 + \log 2}{n(\log n - 1)} \\ &\equiv \delta. \end{aligned}$$

By solving for C , we have that if $C \leq (1 - \delta) n(\log n - 1) / \log 2 - 1$ then $\mathbb{P}[T \neq T^*] \geq \delta$ and we prove our claim. \square

A.4 Proof of Theorem 3

Proof. Our first goal is to find the condition for which the path query Q is equal to its noisy approximation Q' . We start by making some observations for a single fixed node pair (i, j) and later extends our observations for several node pairs.

Define the random variable $R(i, j) \equiv \frac{1}{m} \sum_{k=1}^m \tilde{Q}^{(k)}(i, j)$. Recall that we defined the *path query* Q' as follows. Let $Q'(i, j) = 1$ if $R(i, j) > 1/2$, and $Q'(i, j) = 0$ otherwise.

Assume that $Q(i, j) = 1$. By Definition 6, we have that $\tilde{Q}^{(k)}(i, j) = 1$ with probability $1 - \varepsilon$, and $\tilde{Q}^{(k)}(i, j) = 0$ with probability ε . Clearly, $\mathbb{E}[\tilde{Q}^{(k)}(i, j)] = 1 - \varepsilon$ and therefore $\mathbb{E}[R(i, j)] = 1 - \varepsilon$. By using Hoeffding's inequality, we have

$$\begin{aligned} \mathbb{P}[Q(i, j) \neq Q'(i, j)] &= \mathbb{P}[R(i, j) < 1/2] \\ &= \mathbb{P}[R(i, j) - \mathbb{E}[R(i, j)] < 1/2 - \mathbb{E}[R(i, j)]] \\ &= \mathbb{P}[R(i, j) - \mathbb{E}[R(i, j)] < \varepsilon - 1/2] \\ &\leq e^{-2m(1/2 - \varepsilon)^2}. \end{aligned}$$

Now assume that $Q(i, j) = 0$. By Definition 6, we have that $\tilde{Q}^{(k)}(i, j) = 0$ with probability $1 - \varepsilon$, and $\tilde{Q}^{(k)}(i, j) = 1$ with probability ε . Clearly, $\mathbb{E}[\tilde{Q}^{(k)}(i, j)] = \varepsilon$ and therefore $\mathbb{E}[R(i, j)] = \varepsilon$. By using Hoeffding's inequality, we have

$$\begin{aligned} \mathbb{P}[Q(i, j) \neq Q'(i, j)] &= \mathbb{P}[R(i, j) > 1/2] \\ &= \mathbb{P}[R(i, j) - \mathbb{E}[R(i, j)] > 1/2 - \mathbb{E}[R(i, j)]] \\ &= \mathbb{P}[R(i, j) - \mathbb{E}[R(i, j)] > 1/2 - \varepsilon] \\ &\leq e^{-2m(1/2 - \varepsilon)^2}. \end{aligned}$$

Assume that Algorithm 1 in Line 3 of Algorithm 7 makes C path queries. That is, assume that queries were made for C node pairs $(i_1, j_1), (i_2, j_2), \dots, (i_C, j_C)$. By the union bound and the previous observations, we have:

$$\begin{aligned} \mathbb{P}[(\exists k = 1, \dots, C) Q(i_k, j_k) \neq Q'(i_k, j_k)] &\leq C e^{-2m(1/2 - \varepsilon)^2} \\ &\equiv \delta/2. \end{aligned}$$

By solving for m , we have that if $m \geq \frac{1}{2(1/2 - \varepsilon)^2} (\log C + \log \frac{2}{\delta})$ then

$$\mathbb{P}[(\forall k = 1, \dots, C) Q(i_k, j_k) = Q'(i_k, j_k)] \geq 1 - \delta/2.$$

That is, with probability at least $1 - \delta/2$, the path query Q is equal to its noisy approximation Q' and thus Algorithm 7 reconstructs the tree T correctly, provided that m is large enough.

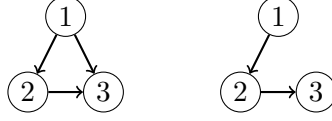


Figure 5: Two directed acyclic graphs that produce the same answers when using path queries.

By Theorem 1, with probability at least $1 - \delta/2$, we have that $C \in \mathcal{O}(\frac{2}{\delta} dn \log^2 n)$. From the above, we have that $m \in \Theta(\frac{1}{(1/2-\varepsilon)^2} (\log d + \log n + \log \frac{1}{\delta}))$ fulfills $m \geq \frac{1}{2(1/2-\varepsilon)^2} (\log C + \log \frac{2}{\delta})$. Finally, the total number of queries is given by $Cm \in \mathcal{O}(\frac{1}{\delta} \frac{1}{(1/2-\varepsilon)^2} dn \log^2 n (\log d + \log n + \log \frac{1}{\delta}))$ and we prove our claim. \square

A.5 Proof of Theorem 4

Proof. Straightforwardly, by the fact that the most time-consuming step in Algorithm 8 is the call to Algorithm 1 in Line 3, and by the result in Theorem 1. \square

A.6 Proof of Theorem 5

Proof. Let \mathcal{G}_n be the set of directed acyclic graphs of n nodes. Next, we show that $\log |\mathcal{G}_n| \in \Theta(n^2)$. For a lower bound, note that if we take any undirected graph $G' = (V, E')$, we can produce a directed acyclic graph $G = (V, E)$ by the rule $(i, j) \in E \Leftrightarrow (i, j) \in E' \wedge i < j$. Given this, the number of directed acyclic graphs is greater than the number of undirected graphs of n nodes, which is $2^{\binom{n}{2}}$. We have $\log |\mathcal{G}_n| \geq \log 2^{\binom{n}{2}} \geq \frac{\log 2}{4} n^2$ for $n \geq 2$ and therefore $\log |\mathcal{G}_n| \in \Omega(n^2)$. For an upper bound, note that the number of directed acyclic graphs is less than the number of directed graphs of n nodes, which is $3^{\binom{n}{2}}$. This follows from the fact that for any two nodes i and j , we have three cases in a directed graph $G = (V, E)$. Either $(i, j) \in E$, or $(j, i) \in E$ or $\{(i, j), (j, i)\} \not\subset E$. Now, we have $\log |\mathcal{G}_n| \leq \log 3^{\binom{n}{2}} \leq \frac{\log 3}{2} n^2$ and therefore $\log |\mathcal{G}_n| \in \mathcal{O}(n^2)$. From the above, we conclude that $\log |\mathcal{G}_n| \in \Theta(n^2)$.

The proof then proceeds as in Theorem 2. \square

The above proof does not take into account the fact that some directed acyclic graphs are non-identifiable by using path queries. For instance, consider the two graphs shown in Figure 5. In both cases, we have that $Q(1, 2) = Q(2, 3) = Q(1, 3) = 1$. Thus, by using path queries, it is impossible to discern whether the edge $(1, 3)$ exists or not. Next, we provide an alternative proof that takes this issue into account.

Proof. Let \mathcal{G}_n be the set of directed acyclic graphs of n nodes. Next, we show that $\log |\mathcal{G}_n| \in \Theta(n^2)$. For a lower bound, we focus on a particular class of “two-layered” graphs. First, the node set $V = \{1, \dots, n\}$ is partitioned into two sets $V_1 = \{1, \dots, \lfloor n/2 \rfloor\}$ and $V_2 = \{\lfloor n/2 \rfloor + 1, \dots, n\}$, known by the graph reconstruction algorithm. We then allow only for edges from nodes in V_1 to nodes in V_2 . That is, each node in V_2 can have as parents any subset of the nodes in V_1 , thus, there are $2^{|V_1|}$ choices of edge sets for each of the $|V_2|$ nodes in V_2 . We have $\log |\mathcal{G}_n| \geq \log 2^{|V_1||V_2|} = \log 2^{\lfloor n/2 \rfloor (n - \lfloor n/2 \rfloor)} \geq \log 2^{n(n-1)/4} \geq \frac{\log 2}{5} n^2$ for $n \geq 5$ and therefore $\log |\mathcal{G}_n| \in \Omega(n^2)$. For an upper bound, note that the number of directed acyclic graphs is less than the number of directed graphs of n nodes, which is $3^{\binom{n}{2}}$. This follows from the fact that for any two nodes i and j , we have three cases in a directed graph $G = (V, E)$. Either $(i, j) \in E$, or $(j, i) \in E$ or $\{(i, j), (j, i)\} \not\subset E$. Now, we have $\log |\mathcal{G}_n| \leq \log 3^{\binom{n}{2}} \leq \frac{\log 3}{2} n^2$ and therefore $\log |\mathcal{G}_n| \in \mathcal{O}(n^2)$. From the above, we conclude that $\log |\mathcal{G}_n| \in \Theta(n^2)$.

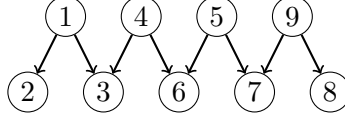


Figure 6: A “v-structured two-layered” directed acyclic graph.

The proof then proceeds as in Theorem 2. \square

A.7 Proof of Theorem 6

Proof. The proof relies on constructing a family of graphs with a single edge. Assume two fixed nodes $i, j \in V$, *unknown* by the graph reconstruction algorithm. The directed graph to be reconstructed is $G = (V, E)$ where $E = \{(i, j)\}$. Note that $Q(i, j) = 1$. Furthermore $Q(k, l) = 0$ for every node pair $(k, l) \neq (i, j)$. That is only one query returns 1, while $n^2 - 1$ queries return 0. Thus, a deterministic algorithm does not obtain any information from the $n^2 - 1$ queries in order to guess the edge (i, j) , and therefore it requires at least n^2 queries in the worst case. \square

A.8 Proof of Theorem 7

Proof. The proof relies on constructing a family of “v-structured two-layered” graphs. Assume the node set V is partitioned into two fixed sets V_1 and V_2 , *unknown* by the graph reconstruction algorithm. For simplicity assume that there is an even number of nodes, and that $|V_2| = |V_1| + 1$. We then create the graph $G = (V, E)$ with $n - 1$ edges such that each node in V_1 is the source of at most 2 edges, and each node in V_2 is the target of at most 2 edges. This creates a “v-structured two-layered” graph as shown in Figure 6. Note that $Q(i, j) = 1$ for $(i, j) \in E$, while $Q(i, j) = 0$ for $(i, j) \notin E$. That is only $n - 1$ queries return 1, while $n^2 - n + 1$ queries return 0. Thus, a deterministic algorithm does not obtain any information from the $n^2 - n + 1$ queries in order to guess the edge set E , and therefore it requires at least $n^2 - n + 2$ queries in the worst case. (Since the algorithm knows that there are $n - 1$ edges, it can stop asking queries as soon as the first bit 1 is returned.) \square

B Experiments

In this section, we present our experimental validation. For each repetition in our experiments, we generate a random bounded-degree directed rooted tree, in order to test whether our Algorithm 1 can successfully recover the tree by using path queries, as well as to count the number of queries needed in practice. We experimentally found that all trees were successfully recovered in practice. Thus, we focused on the question regarding the number of queries needed by Algorithm 1 in practice.

Figure 7(a) shows the number of queries for different number of nodes, while Figure 7(b) shows the number of queries for different node degrees. From our results, it can be observed that the number of queries used by Algorithm 1 is less than what Theorem 1 predicted. Furthermore, for a constant degree d , the experimental results are much better for a large number of nodes n , as shown in Figure 7(a).

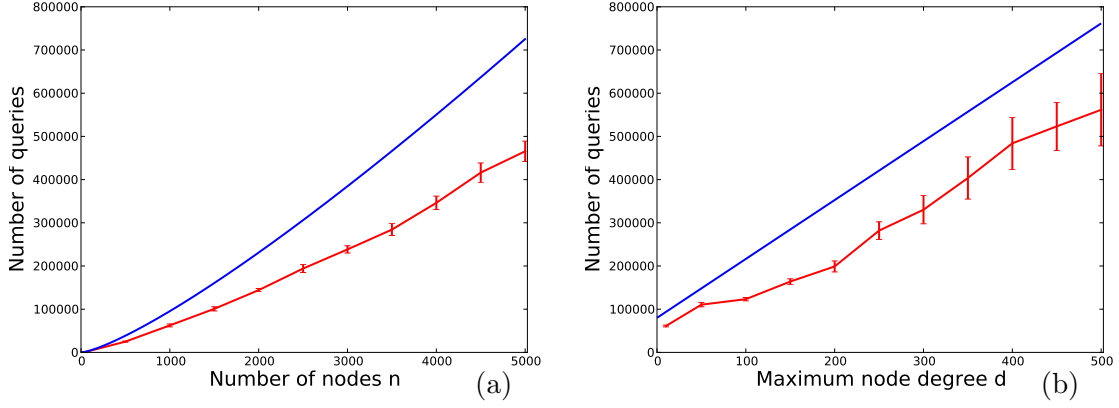


Figure 7: (a) Number of queries for different number of nodes n , for a maximum node degree $d = 5$. (b) Number of queries for different node degrees d , for a number of nodes $n = 1000$. The upper bound for the number of queries in Theorem 1 is shown in blue. The actual number of queries used by Algorithm 1 is shown in red. (Error bars were computed for 10 repetitions, at 95% significance level.)